Diogo Marques Lima

Licenciado em Engenharia Informática

Conceção e implementação de um ambiente de baixo custo para processamento de imagens tomográficas

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador : Prof. Doutor Pedro Abílio Duarte de Medeiros, Prof. Associado, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Pedro Barahona

Arguente: Prof. Doutor Salvador Pinto de Abreu

Vogal: Prof. Doutor Pedro D. Medeiros

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2013

Conceção e implementação de um ambiente de baixo custo para processamento de imagens tomográficas

Resumo

O projeto Tomo-GPU apoiado pela FCT / MCTES está centrado no desenvolvimento de um ambiente gráfico interativo, que ajuda os investigadores da área de Ciências dos Materiais a analisar imagens tomográficas de materiais compostos. Assim, o Tomo-GPU auxilia a identificação e caracterização dos objetos das imagens tomográficas, permitindo ao utilizador determinar critérios de pesquisa como dimensões, orientação, forma, etc.

O hardware a que o software em desenvolvimento se destina é composto por um PC de secretária equipado com um ou mais GPUs. O GPU é usado para efetuar a maior parte das operações de processamento de imagem a três dimensões, o que evita o acesso a clusters remotos com tempos de resposta incompatíveis com uma utilização interativa. Recentemente, foi introduzido no mercado uma nova arquitetura de processadores que contêm no mesmo chip um CPU com múltiplos núcleos (cores) e um GPU – na terminologia do fabricante AMD/ATI estes componentes são chamados APUs (Accelerated Processing Unit)s. Esta nova arquitetura alcançou um grande sucesso em portáteis e dispositivos móveis porque permite reduzir o número de componentes e o custo.

Assim, o objetivo da presente dissertação é o desenvolvimento e otimização de algoritmos de processamento de imagens tomográficas para APUs, e a comparação dos tempos de execução obtidos com os conseguidos por GPUs mais dispendiosos. Para o efeito, será utilizado o ambiente OpenCL uma solução heterogénea que funciona tanto em CPU como em GPU.

Em caso de sucesso desta experiência, os investigadores de Ciência de Materiais terão a possibilidade de utilizar o Tomo-GPU numa plataforma hardware

de custo muito baixo. As experiência até agora realizadas sugerem que os tempos de execução em APUs de algoritmos simples não são competetivas com os conseguidos com GPUs de topo de gama da nVidia.

Abstract

The Tomo-GPU project supported by FCT / MCTES, aims at buiding an interactive graphical environment that help Materials scientists to analyze tomographic images of composite materials. Tomo-GPU helps to identify and characterize objects in tomographic images, regarding dimensions, orientation, shape, etc.

Currently, the project is developing software that is targeted at desktop PCs that include one or more Graphical Processing Units (GPUs); the GPUs are used to perform compute-intensive 3D image processing operations, replacing computing clusters that have turnaround times incompatible with an interactive use of the environment. Recently, a new family of processors has been put in the market characterized by the integration in the same chip of multiple classical CPU-cores and a GPU - AMD calls this processor architecture Accelerated Processing Unis (APUs). This new type of processor has been successfully used in laptops and mobile devices where it allows a reduction in chip count and price.

Therefore, the thesis main goal is to develop and optimize algorithms for processing tomographic images targeted for APUs , and compare the execution times achieved with those of more powerful separate GPUs. The development will use the framework OpenCL (Open Computing Language) that is the industry standard for heterogeneous computing, allowing the combined use of the CPU and GPU parts of the APU.

If the experience succeeds, Materials Science researchers will have the possibility of running Tomo-GPU in less expensive hardware platforms. The experiments made suggest that execution times of implementations of simple image processing algorithms in APUs are not competitive with those obtained in state

of the art nVidia GPUs.

Conteúdo

1

Contexto

1.1 Projeto Tomo-GPU

O projeto FCT/MCTES (PTDC/EIA-EIA/102579/2008 – Ambiente de Resolução de Problemas para Caracterização Estrutural de Materiais por Tomografia – daqui para frente designado por Tomo-GPU – tem como principal objetivo a criação de uma plataforma computacional que ajude os investigadores de Ciências dos Materiais a analisarem imagens tomográficas de materiais compostos.

Os materiais compostos são constituídos por uma base ou matriz na qual se difundem partículas ou reforços de um outro material; o composto assim obtido tem propriedades que nenhum dos dois componentes isolados conseguiria. Este tipo de materiais é cada vez mais importante para um conjunto alargado de indústrias, pelo que estão constantemente a surgir novos métodos para os produzir.

Uma das formas de avaliar as características de um material composto é obter imagens tomográficas de uma amostra e fazer a caracterização da população de reforços, isto é, descobrir informação sobre cada partícula (volume, área, orientação, etc) e também sobre como é que população de reforços se distribui pela amostra.

As imagens tomográficas correspondem a uma matriz a três dimensões em que cada elemento da matriz (voxel) representa a zona da amostra com as coordenadas correspondentes; o valor associado a um voxel é um inteiro não negativo

correspondendo à maior ou menor absorção da radiação naquela parte da amostra. Numa imagem ideal existiriam zonas brancas – correspondentes à matriz – e partes pretas correspondentes aos reforços.

Devido aos processos usados para obtenção da imagem, esta apresenta vários defeitos que é preciso remover antes de se proceder à sua análise. Por outro lado, se a densidade dos materiais que constituem a matriz e os reforços não for muito diferente, é bastante difícil conseguir distingui-los na imagem. Em resumo, antes de se poder fazer a caracterização das partículas é preciso fazer um conjunto de operações sobre a imagem. Essas operações variam de acordo com o tipo da imagem e o seu êxito depende bastante de um conjunto de parâmetros. Neste aspeto, é essencial que um especialista de Materiais possa ir observando o resultado das operações e ajustar os parâmetros de forma a conseguir os objetivos pretendidos. Uma versão deste ambiente está descrita em [**Cadavez2010**].

Do ponto de vista computacional, grande parte das operações acima referidas correspondem à realização de operações sobre os elementos de uma matriz a três dimensões. Muitas das manipulações efetuadas envolvem algoritmos complexos e grandes volumes de dados, o que implica a disponibilidade de um elevado poder computacional.

A plataforma que o projeto Tomo-GPU está a desenvolver tem as seguintes características:

1. **Facilidade de uso** a configuração dum problema na plataforma é simples de realizar mesmo para um especialista em Materiais.

2. **Flexibilidade** a adição ou modificação de novas operações de tratamento é fácil.

3. **Interatividade** mesmo quando se aplicam operações complexas a grandes volumes de dados, os tempos de resposta devem permitir que os utilizadores interatuem com o sistema, podendo mudar os parâmetros do problema facilmente.

4. **Baixo custo** o sistema deve correr num computador de secretária equipado com um controlador gráfico de uso geral (GPGPU). Por outro lado, o software utilizado deve ser não-proprietário e portanto, tendencialmente gratuito.

Para atingir os objetivos acima descritos, o projeto Tomo-GPU baseou-se até agora na utilização do toolkit SCIRun para a parte de interação com o utilizador

e no uso de GPUs para assegurar baixos tempos de execução nas operações de processamento das imagens tomográficas.

O SCIRun [**SCIRun**] é um toolkit vocacionado para o desenvolvimento de "Problem Solving Environments", isto é ambientes interactivos dedicados à resolução de um problema numa dada área da Ciência ou da Engenharia e que são usados por especialistas da área. Tem disponíveis facilidades de visualização de dados científicos e baseia-se em programação visual. Assim, a construção de um programa é feita através da escolha de componentes num menu que depois se interligam através de canais unidirecionais (ver figura 1.1).

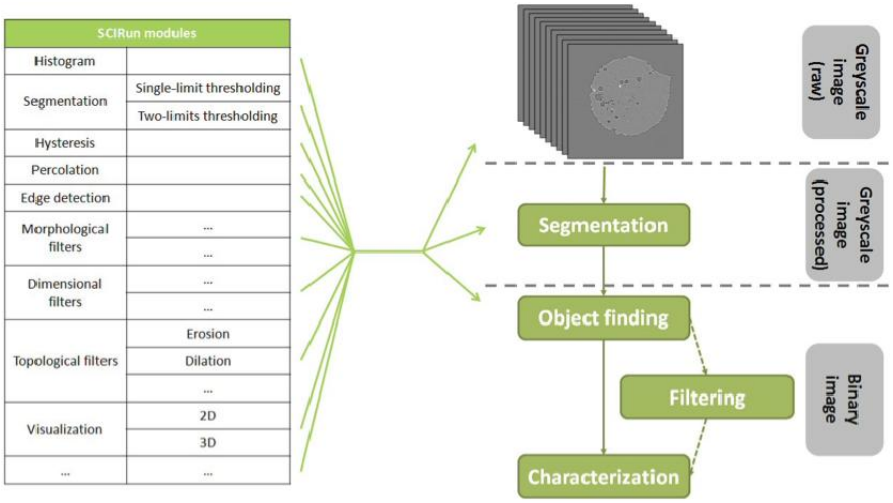

Figura 1.1: Esquema geral do projeto Tomo-GPU [**Velhinho2012**]

No processamento de uma imagem tomográfica são usados alguns componentes já existentes no SCIRun, mas a maioria dos módulos usados foi desenvolvida especialmente para o projeto. Para garantir tempos de execução compatíveis com a interatividade, muitos dos módulos usam as capacidades de processamento de GPUs, sendo utilizada a Framework OpenCL [**Gaster2011**] no seu desenvolvimento.

1.2 Trabalho a realizar

Seria interessante que o Tomo-GPU pudesse correr em plataformas hardware de custo mais baixo do que a actual. O componente que mais pesa no custo é um GPU; os custos de um GPU andam entre os 500 e os 2000 euros.

O objetivo principal desta dissertação é o de estudar a viabilidade de utilizar

um tipo de processador heterogéneo conhecido como APU (accelerated processing unit) para executar algoritmos para processamento de imagens tomográficas. Um APU [**APU**] é constituída por um CPU e um GPU no mesmo chip (on-chip), e que partilham memória. Embora a principal razão para o aparecimento dos APUs seja o seu uso em portáteis e dispositivos móveis, existem versões com CPUs e GPUs relativamente poderosos, nomeadamente os oferecidos pelo fabricante AMD.

Se o APU conseguir executar os algoritmos de processamento de imagens tomográficas em tempos compatíveis com os exigidos pelos utilizadores da plataforma Tomo-GPU, esta poderá ser disponibilizada em hardware de ainda mais baixo custo.

A APU apresenta propriedades diferentes da versão do GPU discreto, onde o CPU e o GPU estão ligados por um bus PCIe, e portanto elimina alguns problemas que limitam a performance, como por exemplo as transferências de dados no bus. Por outro lado, a capacidade de processamento dos CPUs e GPUs integrados é inferior ao conseguido por elementos separados; existe também a limitação da largura de banda de acesso à memória pelo GPU do APU ser inferior à conseguida pelos GPUs que estão instalados no bus PCIe. Estas questões são discutidas mais em detalhe no capítulo 2.

Do ponto de vista do software, continuará a ser usado o OpenCL que, como plataforma concebida inicialmente para programar configurações heterogéneas, é naturalmente proposta para programar os APUs. Esta situação permitirá usar as implementações já existentes para GPUs nVidia como ponto de partida; seguidamente far-se-á a adaptação ao APU.

Existe já um conjunto alargado de módulos do Tomo-GPU, com características diferentes quando às relações entre a computação efetuada no GPU e o tempo gasto nas transferências de informação entre a memória do CPU e a memória do GPU. O transporte, avaliação e adaptação desses módulos permitirá tirar conclusões sobre o potencial do uso do APU.

Refira-se que o projeto Tomo-GPU adquiriu uma máquina equipada com um APU. Uma conclusão que pode ser já apresentada é a vantagem em termos de custos (600 € por uma máquina completa), quando se compara com o custo de um GPGPU nVidia de topo de gama (2500 € por um GPU Fermi C2070, ou 550 € por um controlador gráfico nVidia GTX680)

1.3 Contribuições

A principal contribuição da tese será uma avaliação da possibilidade do uso de APUs na plataforma hardware de execução do ambiente Tomo-GPU. Caso essa avaliação seja positiva, o software Tomo-GPU poderá ser disponibilizado numa plataforma hardware de custo muito acessível, o que naturalmente permitirá aumentar o número de especialistas da área de Materiais que o poderão usar. Naturalmente, que ter uma maior comunidade de utilizadores é benéfico do ponto de vista do projeto, nomeadamente porque permitirá aumentar os contributos de avaliação dos vários componentes do projeto.

Uma outra contribuição desta tese é o transporte parcial do Tomo-GPU para o sistema Voreen [**Voreen**]. O Voreen é um sistema vocacionado para a visualização de volumes de grandes dimensão e que organiza o processamento da informação através de uma rede de módulos. O sistema fornece um conjunto grande de módulos, mas permite a integração de outros desenvolvidos pelo utilizador. O Voreen é uma alternativa interessante ao SCIRun por, nomeadamente:

- Integra facilmente módulos em que computações pesadas são efectuadas em GPUs (quer CUDA quer OpenCL).
- Utiliza bibliotecas mais recentes e melhor suportadas (Qt)
- Tem uma comunidade de suporte muito activa.

1.4 Organização do texto

O capítulo 2 deste texto faz-se um resumo das principais características dos GPUs do ponto de vista hardware; conclui-se com uma súmula das ferramentas usadas no desenvolvimento de aplicações para GPUs. No capítulo 3 apresenta-se conceção da solução, com a abordagem do toolkit usado, o Voreen e o suporte ao OpenCL; no capítulo 4 descreve-se a implementação dos módulos e algumas experiências realizadas. Por fim o capítulo 5, resume as conclusões do trabalho e aponta o trabalho futuro.

2

Trabalho Relacionado

Os objetivos do trabalho foram delineados na capítulo 1; algumas das técnicas e ferramentas envolvidas no desenvolvimento do ambiente para a análise de imagens tomográficas serão apresentadas neste capítulo. Recorde-se que este sistema de análise tem como principais características.

1. Ser disponibilizado numa plataforma hardware de baixo custo;

2. Ser flexível na definição dos processamentos a efetuar e fácil de usar por não especialistas em Informática;

3. Mesmo em hardware de baixo apresentar níveis de desempenho que sejam compatíveis com o tempos de resposta usuais num computador pessoal;

2.1 Processamento de imagens tomográficas

As imagens tomográficas são representadas por uma matriz a três dimensões (3D) na memória do sistema. Uma parte significativa das operações de processamento de imagens 3D corresponde a computações intensivas que envolvem fazer o mesmo processamento a cada uma dos voxels da imagem 3D. Dado o grande volume de dados (uma imagem típica tem 1024x1024x1024 voxels) e a complexidade de algumas operações que têm que ser aplicadas a cada voxel tornam esta área um campo natural de aplicação de processamento paralelo. Diferentes

configurações hardware (multi-processadores de memória partilhada, clusters e grelhas computacionais) têm sido utilizadas neste contexto.

2.2 Ferramentas para construir aplicações baseadas em componentes

Os vários ambientes para o desenvolvimento de ambientes de resolução de problemas (PSEs) - OpenDX, SCIRun, Voreen - usam o paradigma da criação gráfica de programas através da interligação de módulos disponíveis num menu. Um utilizador pode também interligar esses módulos de forma a obter um pipeline de processamento; para cada módulo é possível definir em cada execução diferentes parâmetros. Esta forma de construir programas cumpre os requisitos enunciados anteriormente de facilidade de uso por especialistas da área de Materiais.

2.3 Aceleração das computações associadas a um módulo

Uma parte significativa das operações de processamento de imagens 3D pode tirar partido de técnicas de computação paralela. No nosso caso, alguns dos módulos de processamento serão programas paralelos.

A aceleração conseguida pelo uso de múltiplos processadores depende das características do problema, dependendo dos seguintes fatores:

- Relação entre o tempo de processamento e tempo gasto a lançar a computação e a transferir a informação necessária;
- Independência entre as computações efetuadas pelos várias elementos de processamento;
- Adequação do modelo de computação ao modo nativo de funcionamento do elemento processador. Por exemplo, alguns processadores têm como modo natural de funcionamento o paralelismo de dados enquanto outros estão mais à vontade em paralelismo de controlo;

O hardware alvo do trabalho conducente a esta dissertação é uma configuração heterogénea, pelo que é necessário ajustar os algoritmos de processamento a um modelo de execução muitas vezes conhecido por *off-loading*. Neste modelo os CPUs convencionais executam parte das operações, enviando para os GPUs partes em que estes são mais eficientes.

2.4 Uso de GPUs para acelerar computações

Nesta secção apresentam-se as principais características dos GPUs do ponto de vista hardware; conclui-se com uma súmula das ferramentas usadas no desenvolvimento de aplicações para GPUs, com ênfase para o OpenCL.

2.4.1 GPU

GPU é a sigla para Graphics Processor Unit, que é um processador especializado para efectuar todos os cálculos relativos à manipulação de objetos gráficos a duas e três dimensões. Diverge de outros processadores pelo número muito elevado de elementos de processamento nele existente. Para termo de comparação, a placa gráfica AMD Radeon 6970 tem 24 cores, enquanto a família de CPU da AMD Phenom II tem no máximo 6 cores.

O GPU suporta em hardware um número muito elevado de fluxos de execução (threads) que executem um pequeno conjunto de instruções, ao passo que num CPU clássico, existe normalmente um número reduzido de threads que executam um número muito grande de instruções. Para entendermos este paradigma é necessário perceber que os GPU foram pensados para manipular objetos gráficos, sendo a unidade de trabalho o cálculo de um pixel.

O GPU aparece nos anos 80, com o surgir dos ambientes gráficos nos computadores pessoais. A Intel introduziu a iSBX 275 Video Graphics usado para desenhar linhas, arcos e retângulos. Em 1985, o Commodore Amiga foi o primeiro PC a usar um GPU, usado para acelerar a manipulação de imagens bitmap. Nos anos 90, aparecem as primeiras API para manipulação de objetos gráficos, o OpenGL[**OpenGL**] (Open Graphics Library) e o DirectX [**DirectX**] com o intuito de facilitar o desenvolvimento de aplicações para o mercado dos PCs.

2.4.2 GPGPU

Durante os anos 90, os investigadores aperceberam-se do poder computacional do GPU e começaram a usar as APIs disponíveis para resolver problemas de carácter científico. Isto era uma tarefa bastante complexa por duas razões, a primeira era que, os investigadores tinham que mapear os cálculos científicos em problemas que pudessem ser representados em polígonos e a segunda, os investigadores teriam de deter o conhecimento das APIs. Foi, assim, a necessidade da criação duma plataforma que facilitasse a programação numa linguagem mais fácil de usar pelos programadores tradicionais. Aqui, nasce um novo conceito que

é o GPGPU.

O acrónimo GPGPU - *General-purpose computing on graphics processing units* - foi introduzido em 2006, na altura do lançamento pela nVidia da *framework* CUDA[**CUDA**]. A expressão "General-purpose computing" significa que se usa uma linguagem próxima de uma linguagem tradicional (C / C++) para especificar as acções a efectuar pelos GPUs. Estes ambientes são mais fáceis de usar para programadores que não conhecem a forma de funcionamento do *pipeline* gráfico.

Atualmente, existem múltiplas propostas para programar GPGPUs sendo as mais importantes CUDA, o OpenCL e o Microsoft DirectCompute.

2.4.3 Caracterísiticas do GPU

Foi dito anteriormente que o GPU se distingue pela capacidade em realizar computação paralela, por suportar eficientemente um número elevado de threads. Estas threads têm o objetivo primário de manipulação de objetos gráficos; um exemplo será, se aplicarmos uma rotação a um cubo, é preciso saber a cor que cada pixel terá, ou seja, a visualização da rotação do cubo é dividida em partes mais pequenas, em que cada parte é representada por um pixel. A arquitetura do GPU é classificada segundo a taxonomia de Flynn como SIMD, Single Instruction Multiple Data, o que significa que a mesma instrução é aplicada a dados elementares distintos.

Outro aspeto importante do GPU, é que os dados residem numa hierarquia de memória complexa, sendo a localização dos dados explicitamente controlada pelo programador. Isto contrasta com a situação normal num CPU em que o hardware e o sistema operativo fazem essa gestão de forma transparente para o programador. Mais detalhes sobre este aspeto serão referidos na secção 2.5 quando se discute o OpenCL.

2.4.4 Localização do GPU na arquitectura de um computador

A maneira tradicional de interligar o CPU e o GPU é através bus PCI-e. A figura 2.3 apresenta esta alternativa [**Gaster2011**].

Como é visível na figura 3, o que está a limitar a performance é o bus PCI Express que apresenta 8 GB/s, muito inferior ao acesso a outros componentes. Por isso, na implementação dum programa deve-se minimizar ao máximo as cópias da memória central (memória RAM) para a memória do GPU e vice-versa. Tipicamente, só deveria haver duas transferências, uma para enviar dados e outra para receber os resultados. Obviamente, depende do programa e também se for

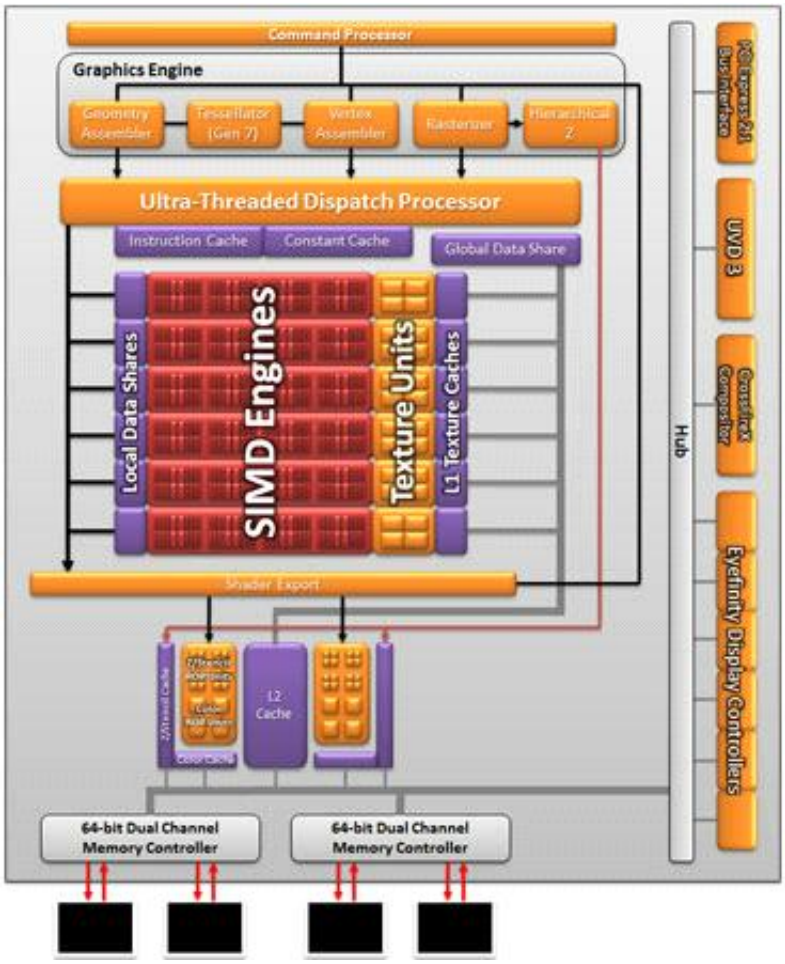

Figura 2.1:

- **SIMD Engines** são as unidades de processamento do GPU;
- **Local Data Share** é a memória partilhada por cada SIMD Unit/core;
- **Texture Units** são responsáveis por receber os dados a serem computados ("fetch" dos dados) e por resolver dependências entre as instruções;

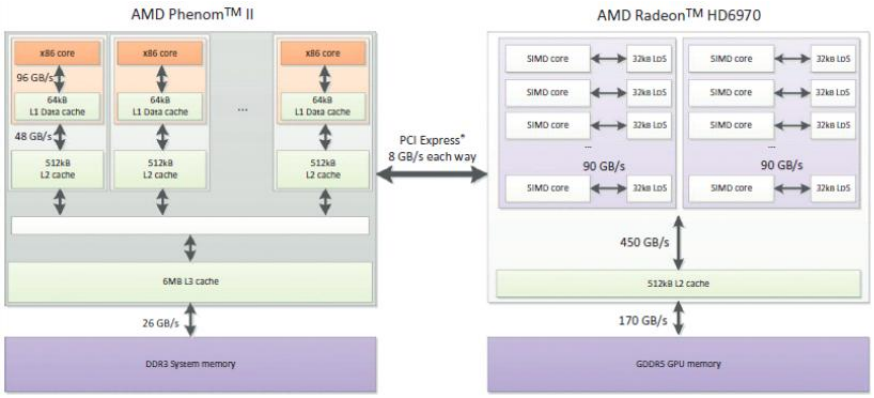

Figura 2.2: O ponto de comunicação entre os dois componentes CPU, à esquerda e GPU, à direita, através dum bus PCIe

apenas utilizado o GPU, mas se forem utilizados os dois ou mais devices (CPU e GPU) para a resolução do mesmo problema, haverá mais dependência de transferências entre os componentes, limitando a performance. Neste sentido, as APUs são mais vantajosas, visto que este "bottleneck" é eliminado.

A versão mais recente do PCI Express é a 3.0 que trabalha a 16 GB/s (Na figura 3, a versão é a 2.0). Em Janeiro de 2012, foi lançada a primeira placa gráfica que suporta PCIe 3.0 [**HD7970**], a AMD Radeon 7970 que para além das melhorias da própria arquitetura (por exemplo, com duas DMA, para enviar e receber dados), nota-se melhorias, não ao nível do processamento gráfico porque, mesmo na versão 2.0 não é utilizada toda a banda disponível, mas sim nesta área do GPGPU, registando-se performances superiores.

Uma boa forma de tentar resolver a questão das transferências será o uso de Zero Copy e Pinned Host Memory, descritas seguidamente.

- **Zero Copy** os dados de entrada são carregados diretamente para a memória do device, e não (como no Copy) carregados para o Host e copiados para o device. A vantagem é que se evita copiar dados.

- **Pinned Host Memory** é uma região de memória que o Host reserva para utilização pelos devices ficando apenas visível para estes, assim o Host não pode usar essa região quando o programa estiver em runtime. O objetivo desta técnica é evitar os page faults por parte doutros processos que estejam a correr no Host, a outra é permitir que a transferência Host para device seja a mais rápida possível.

Apesar das vantagens do Pinned Host Memory, o seu uso nem sempre tem a melhor performance [**AMDTutorial**]. O custo da criação de memória pinned

(pinning) e da libertação (unpinning) é elevado, e no caso dos dados forem superiores a 16 MBytes, são criados buffers intermediários para que o conteúdo seja transferido por parte do Host para o device. Para evitar este problema, é importante que o região tenha A priori os dados a transferir para o device.

2.4.5 APUs

A figura 2.4 apresenta um diagrama de blocos do APU [**APU**].

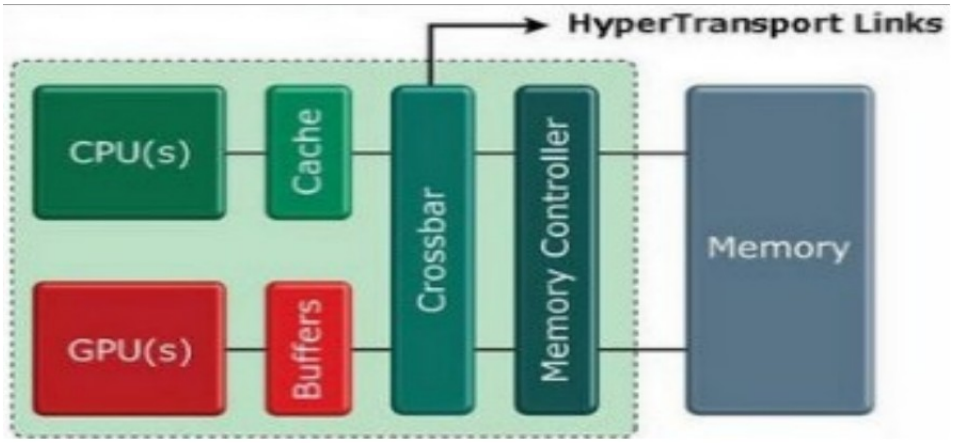

Figura 2.3: Diagrama de blocos do APU

Como se pode observar a interligação entre APU e GPU faz-se através do controlador de memória existindo potencial para conseguir transferências de dados muito mais rápidas entre CPU e GPU. Por outro lado, a taxa de transferência possível entre GPU da APU e à RAM é mais baixa do que aquele que um GPU discreto.

Numa APU, o uso de pinned memory afetará a performance visto que, primeiro não há uso do bus PCIe, segundo, o GPU da APU não tem memória de dispositivo, pertencendo à memória global do sistema. Mais informação sobre a APU será dada mais à frente. EStão em jogo dois efeitos contrários: por um lado o GPU do APU é favorecido porque não há uso do bus PCIe, por outro lado, o GPU da APU não tem memória de dispositivo, tendo a concorrência dos CPUs para acesso memória global do sistema.

2.4.6 Programação em GPUs

Atualmente existem duas formas de programar GPUs:

- Manipulação de objetos gráficos

- Uso de extensões a linguagens "normais" (GPGPU)

Faz-se de seguida uma breve apresentação de alguns dos ambientes usados para programação de GPUs. O OpenCL, usado neste trabalho, é descrito em maior detalhe no capítulo 3.

OpenGL O OpenGL é o acrónimo para Open Graphics Library, que é uma biblioteca com mais de 250 funções para manipulação de objetos gráficos e é gerido pelo Khronos Group. O OpenGL é o ambiente mais usado no desenvolvimento de aplicações gráficas 2D e 3D, incorporando funções para mapeamento de texturas, efeitos especiais, entre outros efeitos. É uma norma disponível em diversas linguagens de programação, diferentes tipos de GPUs da AMD, Nvidia, Intel e S3 Graphics e diferentes sistemas operativos.

Em [**OpenGLOpenCL**] está disponível um tutorial que explica como usar na mesma aplicação o OpenGL e o OpenCL, minimizando as transferências CPU-GPU ao manter os dados na memória do GPU.

DirectX O DirectX [**DirectX**] é tal como o OpenGL, uma API para manipulação de objetos 2D e 3D. Foi desenvolvido pela Microsoft e apenas corre em plataformas da Microsoft como o Windows ou Xbox, é por isso muito utilizado para desenvolvimento de jogos. O DirectX contém vários componentes, como DirectDraw para desenho a 2D, Direct3D o mesmo para 3D, DirectSound, uma interface para a placa de som, DirectInput, ferramenta para controle de dispositivos para jogos como Joystick, entre outros.

A diferença entre o DirectX e o OpenGL, é que o DirectX tem opções para controle de interfaces de hardware, enquanto o OpenGL foca-se mais na manipulação de objetos tendo mais funções para esse efeito. Em suma, o DirectX foi criado para desenvolver jogos e facilitar a sua programação, enquanto o OpenGL para desenvolver aplicações gráficas em geral.

DirectCompute O DirectCompute [**DirectCompute**] é a solução da Microsoft para o GPGPU que está incluído no pacote do DirectX. Pretende competir com as soluções já existentes que são o CUDA e o OpenCL.

CUDA O CUDA é o acrónimo para Compute Unified Device Architecture, é a framework para GPGPU desenvolvida pela Nvidia em 2006 e é o grande concorrente do OpenCL. O CUDA foi pioneiro por ser a primeira Framework para

GPGPU e está implementado em Fortran, C/C++, Java, Haskell, Python, Perl, entre outros.

O problema principal com o CUDA é ser muito limitado no aspeto em que não é multi-plataforma, funcionando apenas em GPUs da Nvidia. Há duas razões para esta limitação, a primeira é óbvia, o CUDA foi implementado pela Nvidia, a segunda foi porque na altura o objetivo era criar uma plataforma programável apenas em GPGPU e não uma solução completamente heterogénea como o OpenCL, que pode trabalhar em simultâneo com o CPU e GPU.

Para reduzir o problema da portabilidade, podemos referir, entre outras, as frameworks Ocelot [**Ocelot**] e OpenACC [**openACC**].

Ocelot O Ocelot permite que os programas CUDA possam ser executados em GPUs da Nvidia e AMD e também em CPU x86. O Ocelot compila os kernels CUDA para as instruções da máquina virtual LLVM que são depois compiladas *Just in Time* para os diferentes tipos de processadores.

OpenACC O OpenACC é um standard para programação em paralelo desenvolvida pela Cray, CAPS, Nvidia e PGI. O objetivo é facilitar a programação em paralelo para programadores não habituados a este género de programação, usando diretivas (# pragma) que permite paralelizar o código. As mesmas diretivas também são usadas na framework OpenMP [**OpenMP**] que só funciona em CPU. Assim o OpenACC procura facilitar a programação de GPUs a programadores já habituados ao OpenMP.

2.5 Dificuldades na programação de GPUs

Como atrás foi visto não faltam ambientes para programar aplicações em GPUs. Apesar desta diversidade, não existem soluções completamente satisfatótias que permitam, em simultâneo:

- Conseguir programas eficientes que consigam apresentar desempenhos semelhantes aos que hardware poderia disponibilizar.
- O permitam fazer a um utilizador que não seja um conhecedor profundo das caracterísiticas de um dado GPU e esteja disposto a perder semanas ou meses a adaptar o seu algoritmo à complexa hierarquia de memória, à organização das unidades de execução e às particularidades do escalonamento hardware dos threads.

3

Conceção da solução

3.1 Visão geral

Os objetivos do trabalho foram delineados na capítulo 1; algumas das técnicas e ferramentas envolvidas no desenvolvimento do ambiente para a análise de imagens tomográficas foram apresentadas no capítulo 2. Recorde-se que este sistema de análise tem como principais características.

- Ser disponibilizado numa plataforma hardware de baixo custo;
- Ser flexível na definição dos processamentos a efetuar e fácil de usar por não especialistas em Informática;
- Mesmo em hardware de baixo custo apresentar níveis de desempenho que sejam compatíveis com o tempos de resposta usuais num computador pessoal;

Como atrás também foi dito, escolheu-se utilizar como plataforma de utilização um computador pessoal de secretária com hardware *off-the-shelf* sendo o único requisito a existência de uma placa gráfica que possa ser utilizada para computações genéricas, suportando o OpenCL como plataforma de desenvolvimento.

Nas secções seguintes começa-se por justificar algumas das opções de implementação feitas, aproveitando-se para descrever dois dos ambientes usados, o

Voreen e o OpenCL. Numa segunda parte do capítulo, descrevem-se algumas fases do processamento de imagens tomográficas, a saber, bi-segmentação, histerese e *labeling* da imagem. As três fases de tratamento descritas vão aparecer no capítulo 4, sendo usadas para comparar o desempenho de um APU com um GPGPU instalado no bus PCIe.

3.1.1 Utilização do ambiente Voreen

Entre os vários ambientes para o desenvolvimento de ambientes de resolução de problemas (PSEs) foi escolhido utilizar o Voreen[**Voreen**]. O Voreen tem uma forma de utilização compatível com outros toolkits do mesmo tipo (OpenDX, SCIRun, ...) usando o paradigma da criação gráfica de programas através da interligação de módulos disponíveis num menu. Um utilizador pode também interligar esses módulos de forma a obter um pipeline de processamento; para cada módulo é possível definir em cada execução diferentes parâmetros. Esta forma de construir programas cumpre os requisitos enunciados anteriormente de facilidade de uso por especialistas da área de Materiais.

O Voreen foi escolhido por apresentar as funcionalidades requeridas, sendo construído a partir de bibliotecas usadas na indústria e estar correntemente em desenvolvimento muito ativo; também a disponibilidade da equipa para responder a questões foi um fator importante na decisão. Outro fator relevante foi a existência de suporte direto para o uso do OpenCL.

3.1.2 Utilização do GPU para off-load de computações

Uma parte significativa das operações de processamento de imagens 3D corresponde a computações intensivas que envolvem fazer o mesmo processamento a cada uma dos voxels da imagem 3D. Neste contexto, é natural que esses processamentos tirem partido da configuração heterogénea existente no hardware, ajustando-se os algoritmos de processamento a um modelo de execução muitas vezes conhecido por off-loading. Neste modelo os CPUs convencionais executam parte das operações, enviando para os GPUs partes em que estes são mais eficientes. O uso do OpenCL permite que o CPU delegue nos GPUs a execução de computações orientadas para o paralelismo dos dados; claro que dada a hierarquia de memória, só faz sentido enviar para o GPU computações em que o tempo de transferência CPU-GPU seja muito menor do que o tempo de processamento no CPU.

3.2 Ambiente de Resolução de Problemas (PSE)

Nesta secção abordam-se o *toolkit* usado para o desenvolvimento do *Problem Solving Environment* para processamento de imagens tomográficas.

3.2.1 Voreen

O Voreen , acrónimo para Volume Rendering Machine é uma framework PSE open source, que permite a visualização interativa de objetos em três dimensões. Desenvolvido em C++, faz uso do GPU e do OpenGL para renderização das amostras. Caracteriza-se por ser uma ferramenta de rápida aprendizagem e bastante flexível na construção e manipulação de redes de visualização. Foi introduzido na Universidade de Münster na Alemanha em 2008. Além disso, o Voreen é multi-plataforma, compatível com Windows, Linux e Mac, e suporta OpenCL, a framework GPGPU usada no projeto e assim a principal razão da sua escolha. Na figura 3.1, apresenta-se um rede Voreen que inclui as várias fases de processamento de uma amostra.

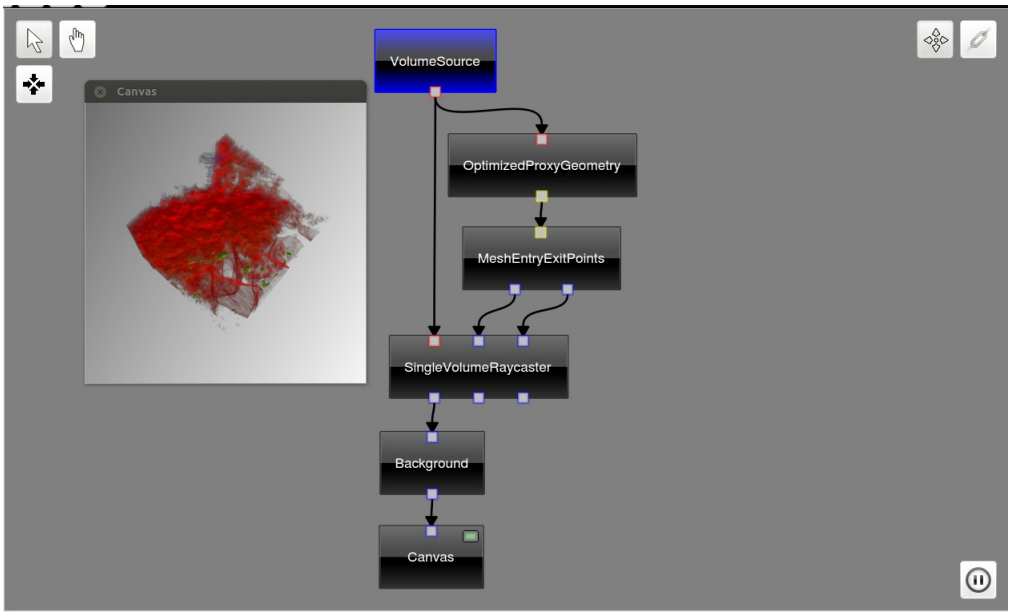

Figura 3.1: Exibição do Voreen em funcionamento

3.2.2 Redes de visualização

A rede de visualização consiste num grafo, onde os nós, os processadores ou módulos, têm portas e entradas e saída (pelo menos uma de entrada ou uma de

saída), que são ligadas numa só direção, essa direção indica o fluxo das transformações realizadas na amostra.

O Processador pode ter diferentes tipos de portas, identificadas com uma cor, no caso do processadores desenvolvidos neste projeto, são apenas usadas portas de tipo volume, que manipulam o conteúdo da amostra, todos os processadores que criam o desenho a três dimensões e aplicam a textura, pertencem à framework do Voreen.

Além disso, o processador pode ter propriedades, permitindo dinamicamente a modificação da amostra. No exemplo das figuras 3.2 e 3.3, verificamos o corte (clipping) na amostra alterando a propriedade do processador, OptimizedProxyGeometry.

3.3 OpenCL

O OpenCL é o acrónimo para Open Computing Language, e é uma Framework GPGPU heterogénea que, para além de correr em GPUs, trabalha em CPUs, DSPs, FPGAs, etc. Foi a adotada pela AMD para a programação dos seus GPUs e APUs. Além disto, o OpenCL é uma especificação aberta ao contrário do CUDA, o que despertou um grande interesse no seio da comunidade e assim tem se desenvolvido bastante. O OpenCL é usado intensivamente neste trabalho e é, por isso, explicado em detalhe de seguida.

Etapas na execução de um programa OpenCL Descrevem-se seguidamente as várias etapas existem na execução de um program que inclui *offload* de computações usando o OpenCL; a esse programa o OpenCL chama kernel. Essas etapas são:

- Preparação da execução
- Preparação dos dados de entrada do kernel
- Lançamento do kernel como um conjunto de threads organizados de acordo com uma grelha a uma, duas ou três dimensões
- recolha dos resultados

Descrevem-se seguidamente alguns dos aspectos envolvidos nestas etapas:

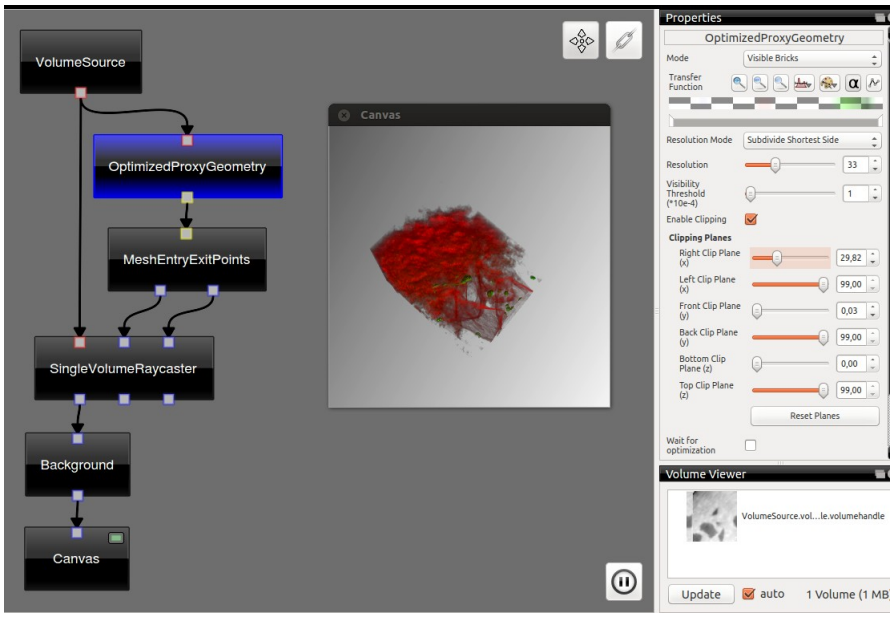

Figura 3.2: Na Coluna direita observamos as diversas propriedades do processador OptimizedProxyGeometry

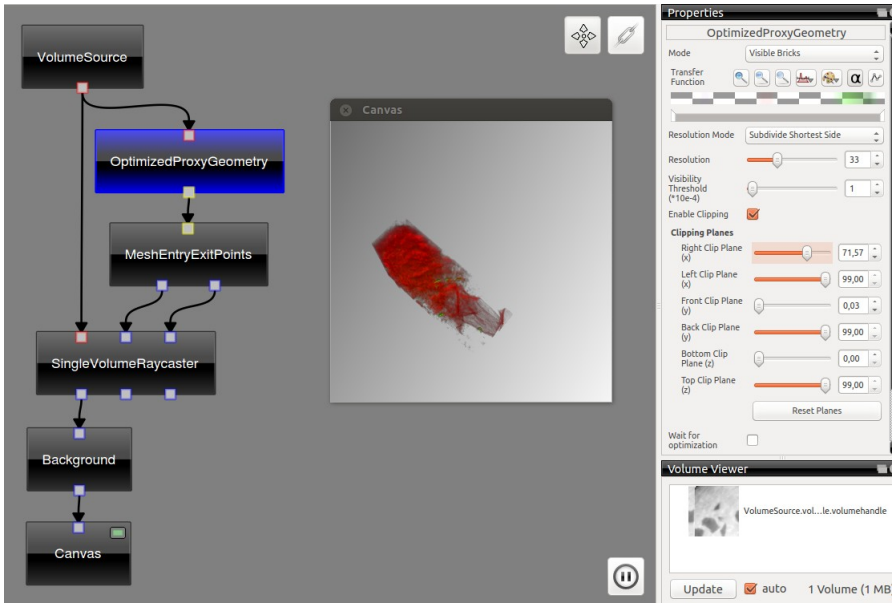

Figura 3.3: A alteração na amostra após a modificação da propriedade

Preparação da execução e dos kernels Como o OpenCL pretende suportar de forma transparente várias plataformas hardware, esta etapa é composta por várias fases:

Criação da plataforma A primeira delas é a criação duma plataforma. Numa plataforma existe um processador (host) capaz de executar toda a parte de controlo do programa e pelo menos um processador (device) capaz de executar o código do OpenCL (kernel). Tipicamente, num computador pessoal,

a plataforma será composta por dois devices (dispositivos), o GPU e o CPU.

Criação do contexto Segue-se a fase da criação do context (contexto) que tem o objetivo coordenar a interação entre o Host e os devices, nomeadamente na gestão dos memory objects (buffers ou imagens), dos programas e da fila de comandos a executar (commandQueue).

Criação de filas de comandos Numa terceira fase, é preciso criar uma ou mais commandQueue em cada device. Assim, a commandQueue tem o papel de concretizar a comunicação entre os Host e devices. Usualmente, são usadas as funções clEnqueueWriteBuffer e clEnqueueReadBuffer, para, respetivamente, transferir dados entre a memória do CPU e a memória do GPU. clEnqueueWriteBuffer transfere dados do CPU para o GPU e clEnqueueReadBuffer do GPU para o CPU. A função que lança a execução de um kernel num dispositivo é clEnqueueNDRangeKernel.

Criação de kernels O último ponto que é a criação dos programas (kernel) e a sua compilação, para que depois ser carregado na commandQueue. Na figura 3.4, encontra-se em termos gerais o fluxo do OpenCL [**AMDTutorial**].

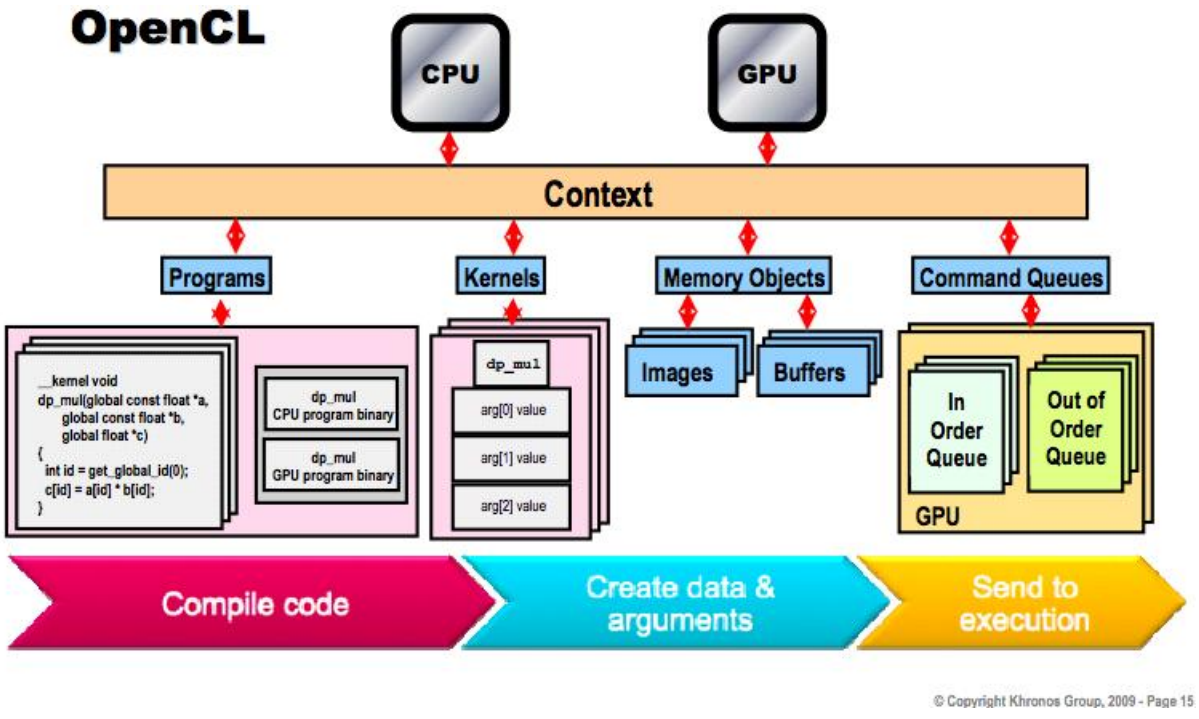

Figura 3.4: O Fluxo do OpenCL., primeira fase tem a compilação do código, seguida pela criação dos objetos de memória e definir os argumentos do programa, e por último a execução da fila de comandos

Modelo de execução associado ao kernel No modelo de execução do OpenCL, um kernel obedece ao modelo SPMD (Single Program Multiple Data), isto é, o mesmo processamento é aplicado a dados diferente. Cada thread criado quando um kernel é chamado *workitem* e obedecem a uma organização hierárquica, sendo agrupados *workgroups*

O *workgroup* é um conjunto de workitems em que tem uma memória local usada para sincronização dos workitems. É lançado ao iniciar a execução e terminado, quando chega ao fim. Quando um workgroup é submetido para execução é criado um thread para cada workitem. O workgroup termina quando todos os threads terminam.

O *workitem* constitui um fluxo de execução do kernel. O que identifica um *workitem* é o seu identificador (id), por isso todos os workitems correm o mesmo código usando o id para fazer acesso a dados distintos.

Vejamos o seguinte kernel, em que cada workitem calcula um elemento do vetor b à custa do elemento homologo da matriz a.

Listing 3.1: Código do kernel

```
1 __kernel void simpleKernel( __global float *a, __global float *b ) {
2   int address = get_global_id(0) + get_global_id(1) * get_global_size(0);
3   b[address] = a[address] * 2;
4 }
```

A função `get_global_id` retorna o identificador global do workitem, o argumento 0 e 1 representa a dimensão deste ("0" é o eixo do xx e "1" é o eixo do yy). O `get_global_size` é o tamanho para a dimensão em argumento. Para além destas primitivas observadas, há também a `get_local_id` que devolve o identificador no workgroup. Na figura 3.5, está esquematizada duas situações possíveis **[Gaster2011]**.

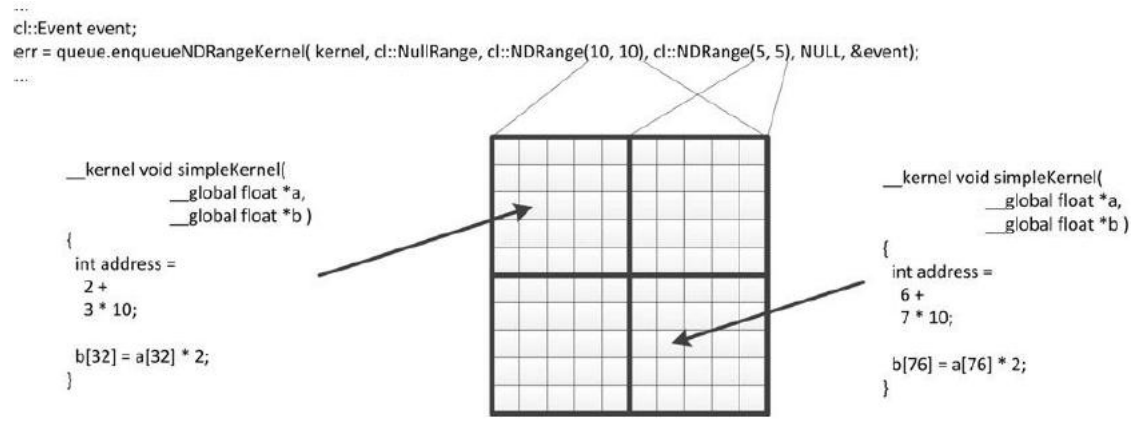

Figura 3.5: A execução do kernel em duas instâncias

Os *workitems* podem sincronizar-se entre si através da memória local do *workgroup*, usando este mecanismo para forçar dependências entre acções. Para isso, há dois modos de sincronização, as barreiras e as *fences*. As barreiras têm um intuito de garantir que todas os threads numa parte do kernel se bloqueiam numa dada instrução I e que só prosseguem quando todas as threads executaram a instrução I. As *fences* estão relacionadas com as escritas em memória, garantindo que os threads só avançam quando todas as escritas na memória efetuadas anteriormente se concluíram. Em termos de desempenho, as fences podem ser consideravelmente melhores porque as threads estão sempre ativas, ao contrário das barreiras que exigem sincronização forçada num certo ponto provocando esperas e além disto podem criar deadlocks.

Modelo de Memória do OpenCL O Modelo de memória do OpenCL por definição é uma abstração da hierarquia da memória que o kernel do OpenCL usa, independentemente da arquitetura do device. O modelo é semelhante ao do GPU, mas pode não ser completamente adaptado a outros aceleradores. Na figura 3.6, encontra-se o esquema do modelo de memória [**AMDTutorial**].

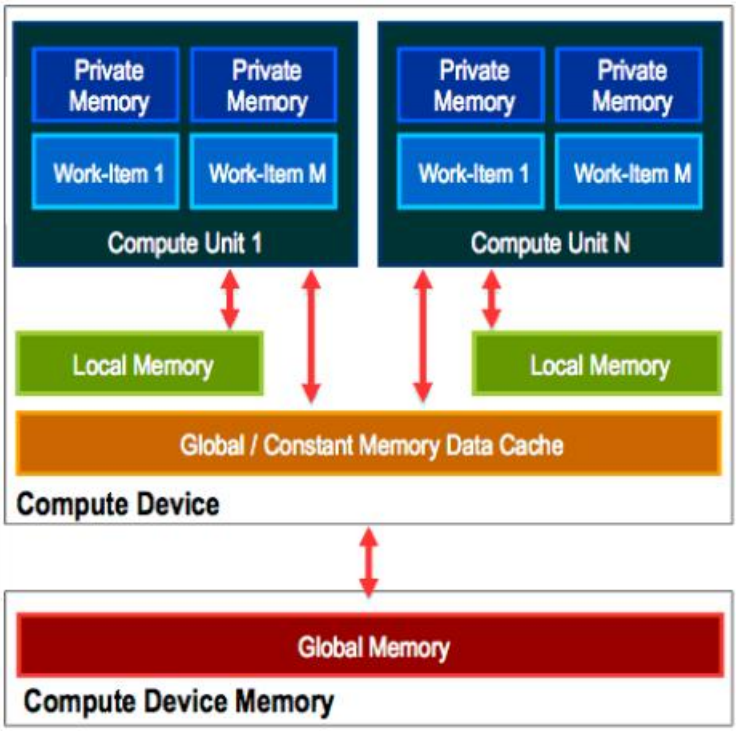

Figura 3.6: Modelo de memória do OpenCL

Há quatro tipos de memória: global, constante, local e privada. No código do kernel, as keywords `__global`, `__constant` e `__local` indicam qual a memória a utilizar, sendo que só o próprio thread pode ter acesso à sua memória

privada.

- **Memória global** é memória com maior capacidade de armazenamento dum dispositivo e é comum a todos os workgroups. O tempo de acesso é o maior de todos e deve-lhe ser feito acesso o menor número de vezes possível. A razão é a latência que envolve o acesso quando comparado com outras memórias, podendo as threads ficar muito tempo em "idle". Por outro lado, o bus de acesso à memória permite a transferência de múltiplas palavras em simultâneo se os acessos feitos pelos threads forem a elementos contíguos (coalescência).

  Outro problema a evitar são os conflitos a bancos de memória, que acontecem quando duas ou mais threads acedem ao mesmo banco de memória, que resulta num abrandamento da performance. Um dos fatores que podem originar este problema está na figura 3.7, são as operações de redução [**Gaster2012**].

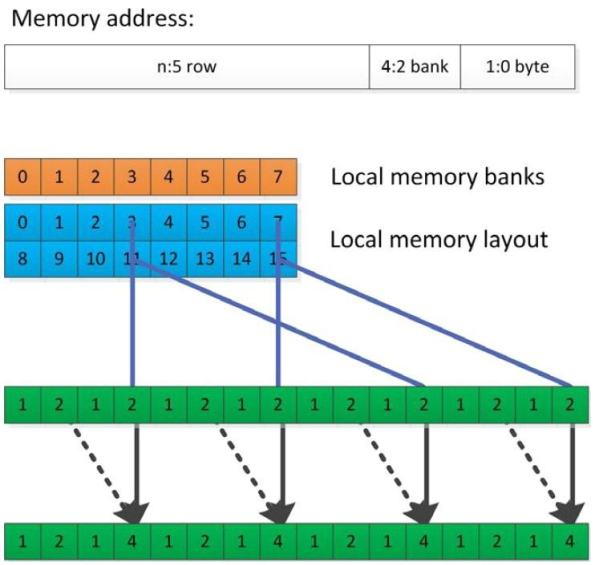

Figura 3.7: A redução está a sempre realizada nos bancos de memória 3 e 7, limitando a performance

- **Memória constante** é a memória "read only" do dispositivo e que é visível por todos os workgroups. Deve ser usada quando existe muitos acessos à mesma região de memória, e é útil para fazer broadcast de dados para todos os threads.

- **Memória local** é a memória rápida de acesso, apenas visível ao workgroup. É utilizada para sincronização de workitems (threads) e para reduzir o número de acessos à memória global.

- **Memória privada** é a memória apenas visível para o workitem, e é a correspondência no GPU aos registos do CPU. É usada para a alocação das variáveis locais no kernel, e por isso toda a alocação feita será em tempo de compilação. Todas as variáveis declaradas num kernel sem qualificativo são privadas ao kernel. Como a memória disponível para os registos é fixa, o número de registos necessários para um kernel limita o número de threads que podem ser lançados em simultâneo.

A transferência de informação entre o *host* e os *devices* utiliza as primitivas *clEnqueueReadBuffer()* e *clEnqueueWriteBuffer()*. Estas primitivas usam as *command queues*. O esquema geral é o seguinte:

- Preparação de argumentos para o kernel:*clSetKernelArg*
- Submissão de buffers com dados para processamento: *clEnqueueWriteBuffer*
- Submissão do kernel e da grelha 1D, 2D, 3D: *clEnqueueNDRangeKernel*
- Aguardar pelos resultados: *clEnqueueReadBuffer*

3.4 Módulos de processamento implementados

No capítulo seguinte apresentam-se três módulos de processamento de imagens tomográficas que foram implementados no Voreen:

Bi-segmentação Este módulo tem como entrada uma matriz 3D com a imagem "em bruto". Na saída estará outra matriz da mesma dimensão em que cada voxel toma apenas três valores: Branco (Matriz), Preto (Reforço), Cinzento (a decidir posteriormente)

Histerese Tem como entrada a matriz produzida pela bi-segmentação e como saída uma matriz em que cada voxel é Preto ou Branco. Os cinzentos são eliminados por um algoritmo iterativo que leva em consideração a maioria das cores dos voxeis vizinhos.

Object labeling Recebe como entrada a saída da fase de histerese e produz uma nova matriz em que cada voxel tem associado um valor inteiro. Voxeis pretos contíguos constituem um reforço. O algoritmo associa a cada conjunto isolado de voxeis pretos uma etiqueta distinta.

Estes três módulos constituem a primeira fase do processamento de imagens tomográficas de amostras de materiais compósitos. Supondo que existe um contraste razoável entre a matriz e os reforços, à saída deste *pipeline* de três módulos teremos, para cada reforço da amostra uma lista dos voxels que a compõem. Esta informação permite passar à fase seguinte do processamento de imagens tomográficas que é obter as características geométricas dos vários reforços. Essas características (dimensões, localização, orientação, etc.) permitem avaliar a qualidade do processo de construção do material compósito em estudo

4

Implementação

Neste capítulo é descrito em detalhe, os módulos implementados no toolkit do Voreen, e o modo como estes se interligam.

4.1 Módulo Bi-segmentação

O Primeiro módulo no processamento de imagens tomográficas deste projeto, é o modulo da Bi-segmentação, descrito na secção 3.4. Assim, o modulo da segmentação tem o intuito de separar a amostra em grupos grosseiramente.

Em baixo, nas figuras 4.1, apresenta-se o mesmo plano da imagem antes da bi-segmentação, e após a bi- segmentação, respetivamente.

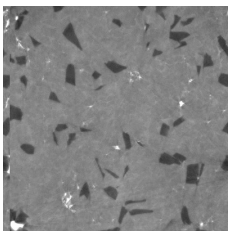

Figura 4.1: Antes da fragmentação

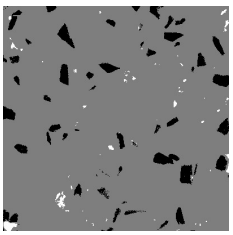

Figura 4.2: Após a fragmentação

Em termos de complexidade, o algoritmo é muito simples, não envolvendo qualquer tipo de cálculo, sendo apenas verificado em que intervalo um determinado ponto se encontra e alterando o seu valor, por forma a pertencer a um grupo. Esse intervalo é manipulado dinamicamente no Voreen, como se observa

na Figura 4.3.

Ao nível do kernel do OpenCL, é usado um apenas um buffer para leitura e escrita, porque todas as operações são independentes de ponto para ponto, ou seja, não há problemas de concorrência, e assim menor alocação de memória. Além disso e também para melhorar a performance, o número de threads iniciadas será "x*y" para amostras a três dimensões (x,y,z), o que significa que cada thread terá que devolver o resultado de "z" pontos, com o objetivo reduzir o custo computacional da criação de threads, o que em amostras de grande dimensão tem impacto; como estamos a fazer dum algoritmo sem cálculos, aumentar o número de computações por ponto não tem peso significativo. Portanto, perder a dimensão da computação paralela, mostrou que tem melhores resultados.

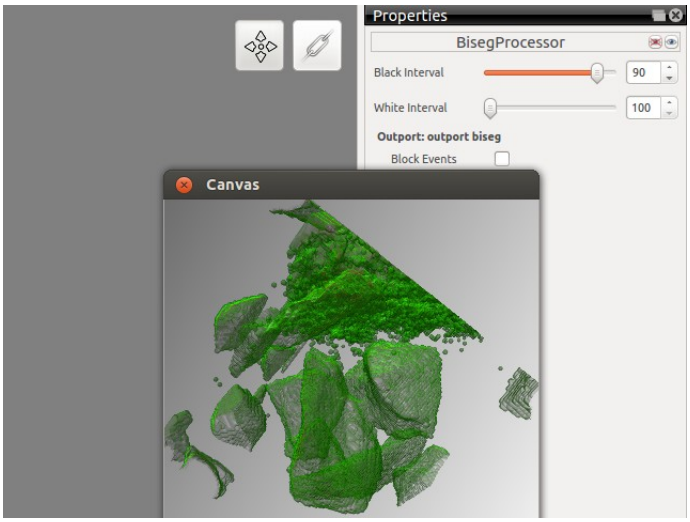

Figura 4.3: Os sliders de intervalo preto e branco. Neste exemplo, observamos que o número de cinzentos está compreendido entre 90 e 100

4.2 Módulo Histerese

O módulo da histerese foi também descrito na secção 3.4. Detalhando, consiste num algoritmo de decisão, onde esta resulta da contagem dos pontos vizinhos de um ponto; o resultado dessa contagem modificará o ponto em análise. Este processo será repetido até se esgotarem todos os pontos cinzentos.

O algoritmo tem duas fases, que são descritas seguidamente.

Algoritmo de maioria O algoritmo da maioria usa a contagem dos pontos vizinhos para a tomada da decisão. Os número de pontos vizinhos são ao todo vinte

e seis, ou seja, vendo cada voxel como um cubo, os seus vizinhos são os voxels enconstados às suas seis faces, oito vértices e doze arestas. O algoritmo é iterativo, querendo isto dizer que o output de uma fase é o input da iteração seguinte.

Para cada voxel, se após a contagem houver maioria relativa de pretos ou brancos, o voxel muda para a cor maioritária; se a maioria relativa for de cinzentos o ponto permanece cinzento. O algoritmo é executado até não haver mais cinzentos, ou até que número de cinzentos não possa ser diminuído.

Algoritmo força bruta No caso do algoritmo de maioria não conseguir eliminar todos os cinzentos passa-se a uma segunda fase que é semelhante à primeira, mas que não considera cinzentos: se há mais vizinhos pretos do que brancos o voxel passa a preto; se há mais brancos passa a branco; se o número for igual escolhe-se aleatoriamente.

Na figura 4.4 e 4.5, visualizamos duas imagens da segmentação e da histerese, respetivamente.

Figura 4.4: Após a segmentação

Figura 4.5: Após a histerese

No exemplo demonstrado nas figuras 4.4. e 4.5, representa uma amostra de qualidade e bem segmentada (intervalos entre 90 e 100), visto que não há um grande número de cinzentos na figura 4.4, logo a histerese aparenta estar bem realizada, como se verifica na figura 4.5. Observemos agora a mesma amostra mas mal segmentada (intervalos entre 90 e 160), verificamos qual o resultado final da histerese, nas figuras 4.6 e 4.7, respetivamente.

Na figura 4.7, a imagem está transformada num bloco preto com algumas zonas brancas, perdendo-se todo o significado da amostra, porque nem toda a zona de preto representam partículas, como se pode verificar na figura 4.5.

4.3 Módulo Object Labelling

No final da histerese, ficamos com dois grupos, em que o preto representam as partículas, assim o próximo passo é identificar cada partícula, subdividindo o

Figura 4.6: Após a segmentação

Figura 4.7: Após a histerese

grupo dos pontos pretos. O algoritmo da identificação é o primeiro a ser submetido, e consiste em etiquetar cada ponto preto no seio de um workgroup. De modo, a explicar melhor todo o processo, observamos a figura 4.8 [**EPCG2012**].

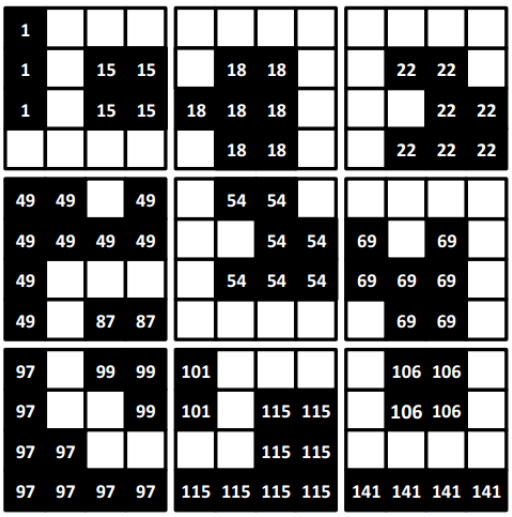

Figura 4.8: Esquema ao fim da execução do algoritmo de identificação

O algoritmo começa por verificar se o voxel em analise, se todos os seus vizinhos são brancos, e em caso afirmativo, um novo identificador é gerado. Os identificadores são gerados em exclusão mútua garantindo que duas partículas distintas não recebem o mesmo identificador. atómicos, significando que representam uma partícula. Nos restantes casos, os voxels recebem o identificador do voxel adjacente com o novo identificador deste.

Na figura 4.8, podemoss observar que apenas dentro do workgroup a separação das partículas é realizada. Uma fase posterior (algoritmo de fusão) faz a união entre sub-partículas em workgroups distintos. A título de exemplo, podemos afirmar que as partículas 89, 99 e 101 são apenas uma partícula.

O processo começa por criar um buffer que tem tamanho igual ao número de identificadores atribuídos anteriormente. Cada posição do vetor representa um identificador e o seu valor, o identificador adjacente com o menor número. Por

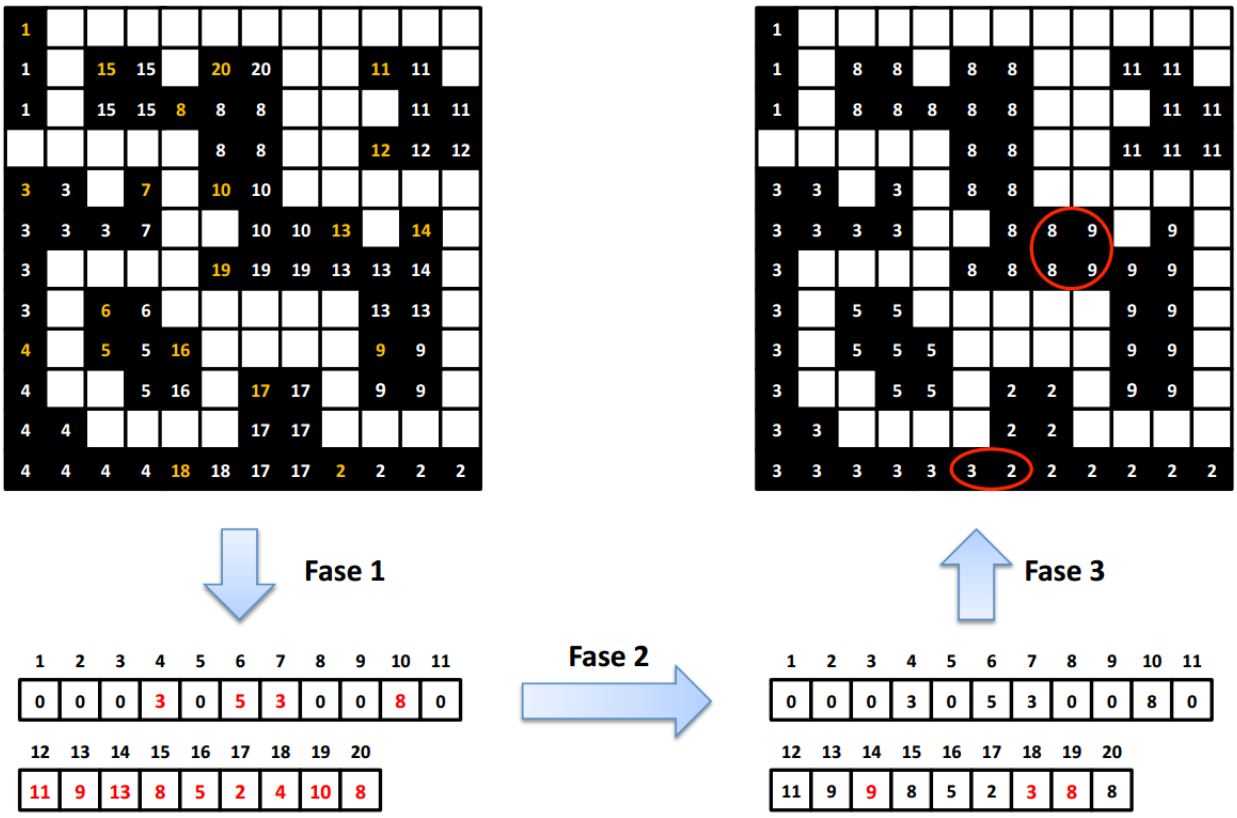

Figura 4.9: A primeira iteração do algoritmo da fusão [**EPCG2012**]

exemplo, a partícula quatro tem duas partículas vizinhas, a partícula três e dezoito, ou seja, o valor na posição quatro é o três. Assim, fica concluída a primeira fase do algoritmo. Importante afirmar que ao nível do OpenCL, é necessário proceder ao sincronismo global de todas a threads, usando a barreira global para o efeito, visto que temos que garantir o correto preenchimento do vetor antes de avançar para as próximas fases.

A segunda fase, funciona apenas na interpretação do vetor, aplicando a regra da transitividade: podemos afirmar que, por exemplo, se o adjacente do quatro é o três e o adjacente do dezoito é o quatro então o adjacente do dezoito é o três. A terceira fase e última, é simplesmente mudar o valor de cada posição no vetor para o valor no ponto.

Terminadas todas as fases, é necessário repetir tudo novamente até que o vetor não sofra alterações em relação à iteração anterior.

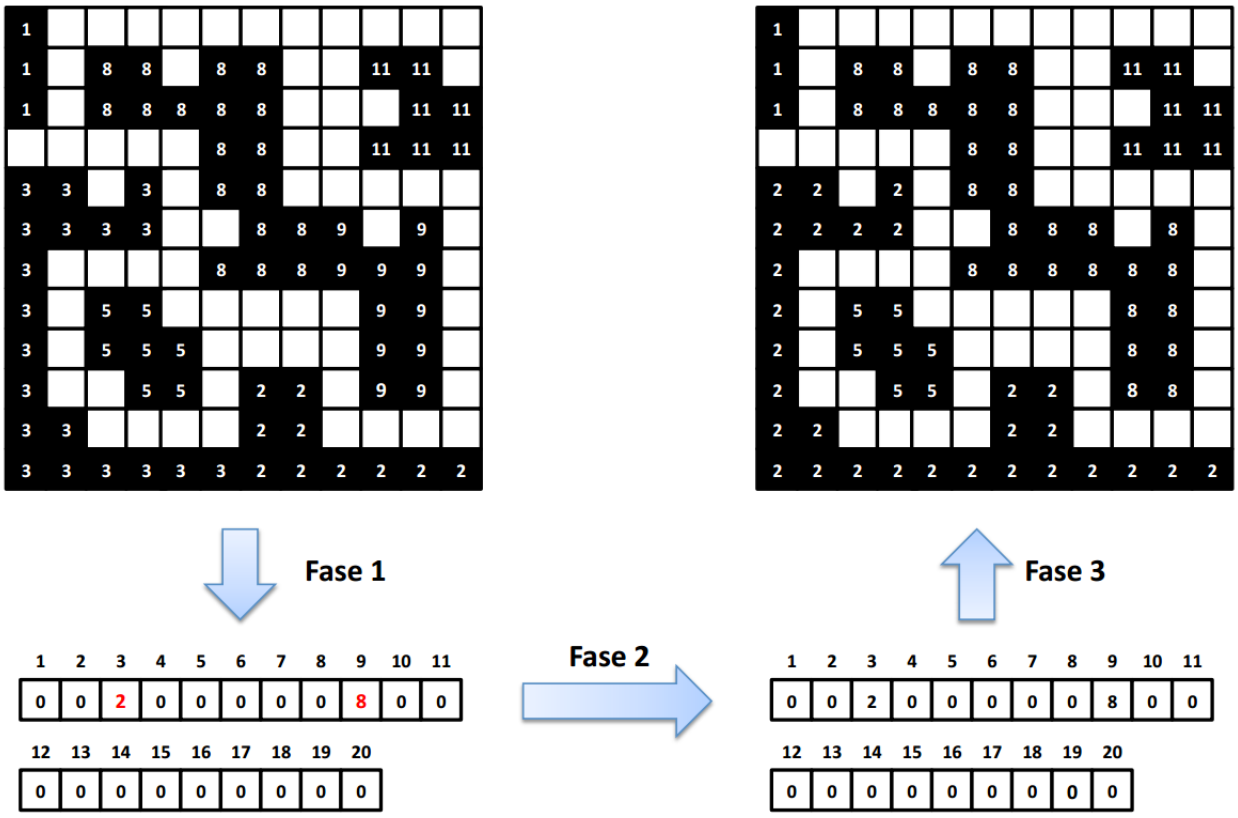

Figura 4.10: A segunda iteração do algoritmo da fusão [**EPCG2012**]

4.4 Resultados de Experiências Realizadas

Na elaboração de resultados das experiências realizadas, foram utilizadas duas máquinas:

Tomo-4 CPU AMD A8-3870 APU with Radeon HD Graphics (4 cores), 8GB RAM, 1 disco SATA de 1 TB

Tomo-2 CPU Intel Xeon E5506 (4-core), 12 GB RAM, 2 discos SATA de 1 TB, um GPU nVidia c2050 Fermi com 3GB RAM (bus PCI-e 16x)

Ambas as máquinas utilizam o Linux Ubuntu 12.04.3 LTS x86_64, kernel 3.2.0-56. No caso do Tomo-4 é usado OpenCL 1.2 AMD-APP e os drivers são *Catalyst 1348.4*; o Tomo-2 usa o OpenCL SDK nVidia 4.2.9 com o OpenCL 1.1 e a versão dos *drivers* é a 295.41.

Os resultados foram obtidos usando apenas os GPUs de cada uma das máquinas, e as amostras têm dimensão 100x100x100 (1000000 voxels), 200x200x200 (8000000 voxels) e 400x400x400 (64000000 voxels). Foi apenas medida a parte relacionada com o OpenCL, não estando incluída a leitura do ficheiro de dados e a escrita do ficheiro de resultados

Bisegmentação		
Tempo (μs)	Tomo-4	Tomo-2
100X100X100	226496	62017
200X200X200	267240	61538
400X400X400	293826	117305

Os resultados mostram a diferença de potencial entre as máquinas *Tomo-2* e *Tomo-4*. O processamento a efectuar no GPU é muito simples, o que permite concluir que a maior parte do tempo gasto está na preparação do contexto e das filas e, no caso das amostras maiores, na transferência dos dados. Um indício desta situação é que o tempo total depende pouco do tamanho das amostras.

Nesta situação, há vantagem teórica da APU nas transferências de/para a memória do GPU; mesmo com essa vantagem o nVidia Fermi tem um desempenho muito superior.

Os resultados para os módulos de histerese e *object labeling* poderão permitir tirar outro tipo de conclusões. As experiências para estes módulos não foram realizadas por dificuldades técnicas; serão, se possível, apresentadas durante a discussão da dissertação.

De qualquer forma, considerando que os algoritmos envolvidos exigem mais poder computacional no GPU é de esperar que a diferença de desempenho ainda seja mais desfavorável ao APU.

5

Conclusões e trabalho futuro

Face aos objetivos enunciados no capítulo 1, este trabalho deu as seguintes contribuições:

- Transporte dos módulos de segmentação, histerese e "Object Labeling" para o sistema Voreen, incluindo o uso do OpenCL e GPU para acelerar as computações mais exigentes do ponto de vista do tempo de execução. Em relação a este transporte foi verificada a possibilidade de transportar o PSE Tomo-GPU para uma plataforma com algumas vantagens em relação ao SCIRun.

- Comparação, num conjunto de algoritmos, do desempenho de um APU com o de um GPU de elevada capacidade e muito maior custo. Deste ponto de vista, as conclusões da dissertação não podem ser consideradas positivas. Os resultados parciais parecem indicar uma grande diferença de desempenho do GPU de um APU com o de um GPU instalado no bus PCI-e; estas diferenças surgem mesmo em situações em que os tempos de trasnferência associados às operações de transferência de dados GPU-CPU deveriam favorecer o APU. Será preciso efectuar mais experiências, mas para já parece ser de concluir que, para operações mais pesadas, não é possível usar APUs sem tornar os tempos de resposta demasiado elevados.

Em termos de trabalho futuro, serão transportados para o sistema desenvolvido nesta tese, outros módulos desenvolvidos no âmbito do projeto Tomo-GPU.

Também se planeia ensaiar outros tipos de hardware para suportar as computações paralelas.